下载APP

04｜控制逻辑：表达式和语句是如何协调程序运行的？

2021-12-13 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 14:40 大小 13.44M

你好，我是于航。今天，我们继续来看 C 基本语法结构背后的实现细节。

上一讲，我主要介绍了编译器是如何使用机器指令来实现各类 C 运算符的。在应用程序的构建过程中，运算符仅作为“计算单元”，为程序提供了基本的“原子”计算能力。而数据如何同时使用多种不同的运算符，以及按照怎样的逻辑来在不同位置上“流动”，这一切都是由表达式和语句进行控制的。这一讲，就让我们来看看 C 语言中，用来描述程序运行逻辑的这两种控制单元“背后的故事”。

表达式

表达式（expression）是由一系列运算符与操作数（operand）组成的一种语法结构。其中，操作数是参与运算符计算的独立单元，也即运算符所操作的对象。操作数可以是一个

简单的字面量值，比如数字 2、字符串 “Hello, world!” ；也可以是另一组复杂的表达式。举个例子：在表达式（1 + 2） * 3 + 4 / 5 中，乘法运算符 “*” 所对应的两个操作数分别是字面量数值 3，和子表达式（1 + 2）。

通常来说，表达式的求值（evaluation）过程需要依据所涉及运算符的优先级和结合性的不同，而按一定顺序进行。我们一起来看看上面提到的（1 + 2） * 3 + 4 / 5 这个表达式的计算流程。

首先，需要根据表达式中运算符优先级的不同，来决定最先进行哪一部分运算。运算符的优先级很好理解，由于乘法运算符 “*” 与除法运算符 “/” 的优先级高于加法运算符 “+” ，因此在计算整个表达式的值时，需要首先对由这两个运算符组成的子表达式进行求值。

当从上一步中“筛选出”的待计算运算符多于 1 个时，我们就需要再判断运算符的结合性，来决定优先计算哪一个。因为乘法运算符和除法运算符均具有左结合性，因此，由左侧乘法运算符构成的子表达式需要被优先求值。

当我们以这个表达式为视角，进行观察时，参与表达式计算的操作数分别为子表达式（1 + 2），以及字面量数值 3。这里，我们需要分别对这两部分进行求值，直至乘法运算符 “*” 两边的操作数可以直接参与计算为止。但需要注意的是，C 标准中并未规定运算符两侧操作数的具体求值顺序，因此具体方式由编译器选择。

因此我们可以说，**对表达式的求值过程，实际上就是根据运算符的优先级和结合性，来对表达式和它所包含的子表达式进行递归求值的过程**。从编译的角度来看，这个过程中所涉及到的操作数的实际求值顺序会在语法分析阶段被确定，并体现在源码对应的抽象语法树（AST，Abstract Syntax Tree）上。为了方便进一步观察，我将这个表达式整合到了下面的 C 代码里，并保存在文件 main.c 中：

复制代码

```
1 int foo(void) {
2   return (1 + 2) * 3 + 4 / 5;
3 }
```

然后，借助 Clang 编译器提供的 “-ast-dump” 选项，我们可以编译并打印出这段 C 代码对应的 AST 结构。完整的编译命令如下：

```
clang -Xclang -ast-dump -fsyntax-only main.c
```

复制代码

上面的命令执行完毕后，部分输出结果如下图所示：

```
`-FunctionDecl 0x5573c3ef30e0 <main.c:1:1, line:3:1> line:1:5 foo 'int (void)'
  `-CompoundStmt 0x5573c3ef3318 <col:15, line:3:1>
    `-ReturnStmt 0x5573c3ef3308 <line:2:3, col:28>
      `-BinaryOperator 0x5573c3ef32e8 <col:10, col:28> 'int' '+'
        |-BinaryOperator 0x5573c3ef3268 <col:10, col:20> 'int' '*'
        | |-ParenExpr 0x5573c3ef3228 <col:10, col:16> 'int'
        | | `-BinaryOperator 0x5573c3ef3208 <col:11, col:15> 'int' '+'
        | |   |-IntegerLiteral 0x5573c3ef31c8 <col:11> 'int' 1
        | |   `-IntegerLiteral 0x5573c3ef31e8 <col:15> 'int' 2
        | `-IntegerLiteral 0x5573c3ef3248 <col:20> 'int' 3
        `-BinaryOperator 0x5573c3ef32c8 <col:24, col:28> 'int' '/'
          |-IntegerLiteral 0x5573c3ef3288 <col:24> 'int' 4
          `-IntegerLiteral 0x5573c3ef32a8 <col:28> 'int' 5
```

AST 作为用于表示源代码语法结构的一种树形数据结构，语法分析器会将表达式中操作数的整体求值顺序映射到树的结构上。因此，当我们以后序遍历（LRD）的方式遍历这棵树时，便可以直接得到正确的表达式求值顺序。

对于上面的 AST 来说，由叶子结点组成的子树需要被最先求值，因此我们首先可以得到括号内加法表达式的计算结果 3。然后，该结果将作为叶子结点上的操作数，参与乘法运算符的计算，从而得到计算结果 9。接下来，除法运算符所在的子表达式经过求值，得到结果 0。最后，该值再作为最后一个加法运算符的操作数，与字面量值 9 相加，进而得到整个表达式的最终计算结果 9。

你可以看到，表达式提供了这样一种能力：**能够让数据同时参与到多个操作符的不同计算过程中**。而通过提供对运算符优先级与结合性的规则限制，表达式可以控制整个计算过程的有序进行。

语句

语句（statement）是用来描述程序的基本构建块。和表达式不同，语句是构成 C 程序的最大粒度单元，在它的内部，可以包含有简单或复杂的表达式结构，但也可以不包含任何内容。除此之外，语句在执行时不返回任何结果，但可能会产生副作用。

在 C 语言中，语句可以被分为复合语句、表达式语句、选择语句、迭代语句、跳转语句五种类型。**但无论是哪种类型，语句都必须以分号结尾，并按从上到下的顺序依次执行。**

其中，复合语句是指由花括号 “{}” 标记的一块区域。在这个区域中，我们可以放置声明（declaration）和语句，而最常见的一种复合语句便是函数体。在函数体内部，我们可以定义变量，并通过结合各类其他语句来实现函数的功能。而表达式语句则是直接由表达式外加一个分号组成的语句，比如函数调用语句或变量赋值语句。当然，表达式语句中的表达式也可以为空，这样就成为了仅由一个 “;” 组成的空语句。

在下面这段代码里，我标注出了其中使用到的复合语句与表达式语句。你可以通过它们来加深对这两种语句的理解。

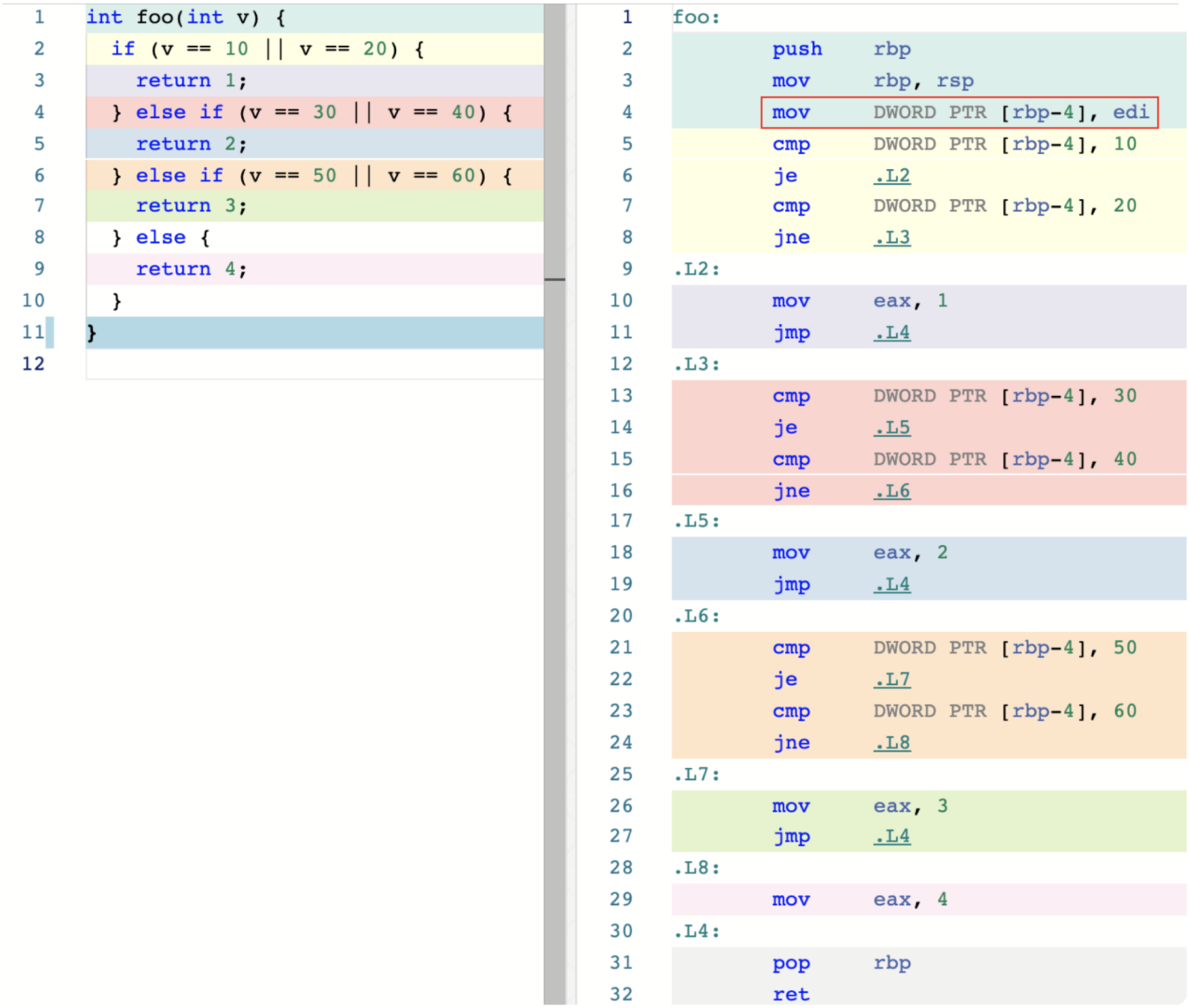
复制代码

```
int foo(int x, int y) {  // 复合语句；
  int sum = x + y; // 表达式语句；
  if (sum < 0) {  // 复合语句；
    sum = -sum;  // 表达式语句；
  }
  return sum;
}
```

这两种类型的语句，它们的具体结构依程序设计细节的不同而不同，因此这里我们不再做更多的讨论。相对的，在 C 语言中，选择语句、迭代语句、跳转语句都有着它们相对应的特定语法结构。因此，接下来我们重点看看这几类语句，探究编译器是如何实现它们的。

选择语句

同其他大多数语言类似，在 C 语言中，选择语句主要是指由 if...else 和 switch...case 这两种语法结构组成的语句。它们的使用方式你应该很熟悉，这里就不多讲了。让我们直接通过一个例子，观察编译器在默认情况下是如何实现它们的。首先来看 if...else 语句：



```
int foo(int v) {
  if (v == 10 || v == 20) {
    return 1;
  } else if (v == 30 || v == 40) {
    return 2;
  } else if (v == 50 || v == 60) {
    return 3;
  } else {
    return 4;
  }
}
```

```
foo:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        cmp     DWORD PTR [rbp-4], 10
        je      .L2
        cmp     DWORD PTR [rbp-4], 20
        jne     .L3
.L2:
        mov     eax, 1
        jmp     .L4
.L3:
        cmp     DWORD PTR [rbp-4], 30
        je      .L5
        cmp     DWORD PTR [rbp-4], 40
        jne     .L6
.L5:
        mov     eax, 2
        jmp     .L4
.L6:
        cmp     DWORD PTR [rbp-4], 50
        je      .L7
        cmp     DWORD PTR [rbp-4], 60
        jne     .L8
.L7:
        mov     eax, 3
        jmp     .L4
.L8:
        mov     eax, 4
.L4:
        pop     rbp
        ret
```

如上图所示，在左侧的 C 函数 foo 中，我们使用 if...else 语句构建了一个简单的程序逻辑。if 语句会在每一个条件分支中检测函数参数 v 的值，并根据匹配情况返回一个数值。若所有情况都没有命中，则最后的 else 语句生效，直接返回数值 4。相应的，在右侧，我们可以看到这个函数对应的汇编代码。

在这里，通过红框内的汇编代码可以看到，变量 v 的值被存放在栈内存中地址为 rbp 寄存器的值减去 4 的位置上。程序使用多个标签（如 .L2、.L3 等），分别划分不同分支对应的处理逻辑，而分支的判断过程则是由指令 `cmp` 与条件跳转指令 `je` 与 `jne` 共同完成的。**汇编代码和 C 代码的整体逻辑基本是一一对应的关系。**因此，在这种情况下，为了尽量保持程序的执行性能，你可以将命中几率较大的条件语句放在较前的位置上。

接着，我们再来看看 switch...case 语句。

C

```
int foo(int v) {
  switch(v) {
    case 10:
    case 20: {
      return 1;
      break;
    }
    case 30:
    case 40: {
      return 2;
      break;
    }
    case 50:
    case 60: {
      return 3;
      break;
    }
    default: return 4;
  }
}
```

ASM

```
foo:
        ...
        mov     eax, DWORD PTR [rbp-4]
        cmp     eax, 60
        seta    dl
        test    dl, dl
        jne     .L2
        mov     edx, 1
        mov     ecx, eax
        sal     rdx, cl
        mov     rax, rdx
        movabs  rdx, 1154047404513689600
        and     rdx, rax
        test    rdx, rdx
        setne   dl
        test    dl, dl
        jne     .L3
        ...
        and     eax, 1049600
        test    rax, rax
        setne   al
        test    al, al
        je      .L2
        mov     eax, 1
        jmp     .L5
        ...
.L3:
        mov     eax, 3
        jmp     .L5
.L2:
        mov     eax, 4
.L5:
        pop     rbp
        ret
```

这里，我只列出了部分汇编代码，但足够你理解编译器对 switch...case 语句的实现细节了。其中，标注为红色的汇编代码会通过 cmp 指令，判断寄存器 eax 中的值，即变量 v 的值是否大于 60。若判断成立，则直接将程序跳转到标签 .L2 处，并将数字 4 作为返回值；若条件不成立，程序将继续执行。接下来，蓝色部分的代码会基于变量 v 的值，来产生一个用于参与后续运算符的 “token” 值。这个值的生成步骤如下所示：

1. 将寄存器 edx 的值设为 1；
2. 将寄存器 ecx 的值设为变量 v 的值；
3. 将寄存器 rdx 中的值左移 v 位（值被扩展为 64 位）；
4. 将此时寄存器 rdx 中的值移动到 rax 中留作待用。

接下来，通过上图中右侧虚线框内的代码，程序完成了对变量 v 的值的第一次筛选过程。这个过程很好理解，如果将其中第一行指令 `movabs` 的立即数操作数 1154047404513689600 以 64 位二进制的形式展开，你会发现其中只有第 50 和 60 位被置位。

而第二行的 `and` 指令，会将这个超长的立即数与之前根据变量 v 的值进行移位而得来的 token 值进行“与”操作。若操作得到的结果不为 0，则表示 token 值的第 50 或 60 位肯定不为 0，即变量 v 的值为 50 或 60。否则，变量 v 的值则不符合该 case 语句的筛选条件。到这里，筛选的基本逻辑相信你已经清楚了。

不过，通过“位映射”的方式进行分支筛选，并不能完美地覆盖所有情况。比如，当 case 语句的筛选值过大，无法使用寄存器来进行映射时，默认优化条件下，编译器会将 switch…case 的实现“回退”到与 if…else 类似的方式。也就是说，使用 `cmp` 指令与条件跳转指令来进行分支的筛选与转移。当然，具体采用哪种实现策略依据编译器的不同而有所不同。

除了上面介绍的 if…else 与 switch…case 语句实现方式外，在高优化等级下，编译器还可能会采用一种名为“跳表”的方式，来实现这两种条件选择语句。

下面是用这种方式修改后的 switch…case 语句实现，你可以先观察下，并思考这种方式的实现思路。

C

```
int foo(int v) {
  switch(v) {
    case 10:
    case 20: {
      return 1;
      break;
    }
    case 30:
    case 40: {
      return 2;
      break;
    }
    default: return 3;
  }
}
```

ASM

```
bar:
        add     edi, -10
        cmp     edi, 30
        ja      .LBB0_3
        mov     eax, 1
        jmp     qword ptr [8*rdi + .LJTI0_0]
.LBB0_2:
        mov     eax, 2
        ret
.LBB0_3:
        mov     eax, 3
.LBB0_4:
        ret
.LJTI0_0:
        .quad   .LBB0_4
        .quad   .LBB0_3
        .quad   .LBB0_3
        .quad   .LBB0_3
        .quad   .LBB0_3
        .quad   .LBB0_3
        .quad   .LBB0_3
        .quad   .LBB0_3
        .quad   .LBB0_3
        .quad   .LBB0_3
        .quad   .LBB0_3
        .quad   .LBB0_4
        .quad   .LBB0_3
        .quad   .LBB0_3
        ...
```

可以看到，这里我们将 switch...case 语句中分支筛选的“跨度”进行了减小，即将其中的最大分支匹配条件由 60 减小到了 40。**跳表是一种用空间换时间的条件匹配策略**，让我们通过上图右侧的汇编代码，来了解它的实现过程。

首先，标注为红色的汇编代码将变量 v 的值减去了选择语句中最小匹配条件的值，这里也就是 10。然后，程序通过 cmp 与 ja 指令，判断变量 v 的值是否超过了选择语句中最大匹配条件与最小匹配条件之间的差值，这里也就是 30。若是，则程序直接跳转到标签 .LBB0_3 处，并返回数值 3。

否则，程序就会使用跳表来寻找变量 v 的值对应的正确分支。**所谓跳表，即在一段连续内存中存放的，可用于辅助查找正确目标地址的地址信息**。在上面这个例子中，跳表从标签 .LJTI0_0 处开始。在这段内存中，连续存放了筛选值 10 到 40 区间内，每一个整数值对应的正确分支处理地址。接下来的蓝色代码保存了当跳表第 0 项“命中”时，函数需要返回的值。

假设在调用函数 foo 时，传入变量 v 的值为 20。虚线框中的 `jmp` 指令在执行时，会根据 v 的值在跳表中找到它所对应的正确分支地址。由于这里 rdi 寄存器中的值为 10（20 - 10），因此正确的分支处理地址便是跳表中第十项对应的值。这里可以看到，在 .LJTI0_0 标签 +80 字节的位置（.quad 代表 8 字节数据）处，正对应着标签 .LBB0_4 的地址。而该标签的位置，正是变量 v 为值 20 时的正确分支处理地址。

除了上面提到的这些编译器在实现分支语句时使用的常用方式外，根据分支语句的具体情况，编译器还可能会采用某些针对特定形态代码的专用优化。而即使针对最“原始”的 `cmp` 加条件跳转语句组合这种实现方式，编译器也会根据 C 源代码的情况，适当使用“二分法”等优化策略，来加快条件的筛选过程。

迭代语句

在 C 语言中，迭代语句主要包含 do...while、for、while 这三种基本语法形式。这些语句除了在执行细节上有些许差异外，其对应的汇编实现思路大同小异。这里我以 do...while 语句为例来讲解，具体代码如下所示：

```c
#include <stdio.h>
void foo(int v) {
  do {
    printf("%d", v);
  } while(v--);
}
```

```asm
.LC0:
        .string "%d"
foo:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
.L2:
        mov     eax, DWORD PTR [rbp-4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, DWORD PTR [rbp-4]
        lea     edx, [rax-1]
        mov     DWORD PTR [rbp-4], edx
        test    eax, eax
        jne     .L2
        nop
        nop
        leave
        ret
```

可以看到，在真正对变量 v 进行条件判断之前，程序已经执行了一次 `printf` 函数，而这便是 do...while 语句相较于其他迭代语句的特点。迭代过程以 .L2 标签作为每次的起始点，每次迭代都遵循着“先执行循环体，再判断条件”的规则。条件的判断和执行转移流程则分别由指令 `test` 与 `jne` 负责进行。

即使是在高优化等级下，C 语言中的这三种基本迭代语句在机器层面的汇编实现方式也不会有较大的差异，但这也并不意味着你可以随意使用它们。至少对于 do...while 与 while 而言，它们在执行细节上存在着差异，如果不假思索地使用，很可能会给你的程序招致不必要且难以调试的 BUG。

跳转语句

C 语言中的跳转语句主要指那些可以改变程序执行流程的语法结构，它们主要包括以下四种类型：

break 语句；

continue 语句；

return 语句；

goto 语句。

其中，return 语句的执行细节涉及到了函数的调用与返回，因此我会在 05 讲中为你详细介绍。而对于另外三种语句，相信就算不参考实际代码，对于它们的实现“套路”，你也已经心中有数，因为它们的基本功能均是改变程序的具体执行流程。

在 C 代码中，用于控制程序执行逻辑的大部分语句，其背后都是通过条件跳转语句来实现的。编译器通过代码分析，可以找到程序中可能的“跳入点”与“跳出点”，并在机器指令层面通过 `je` 等条件跳转指令，来控制程序的执行流程在这些点之间进行转移。也就是说，只要理解了其他语句实现中对条件跳转指令的使用方式，这里你就能融会贯通。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

今天我主要讲解了 C 语言中用于描述程序运行逻辑的两种“控制单元”，即表达式和语句背后的实现细节。

表达式作为一种表达计算的基本语法结构，对它的求值过程需要根据参与运算符的结合性与优先级，按一定顺序进行。而计算的具体顺序则会在语法分析阶段，由编译器直接体现在对应的 AST 结构形态上。

语句是程序的基本构建块，通过不同种类语句的组合使用，我们可以控制程序的执行逻辑。C 语言中的语句主要包括复合语句、表达式语句、选择语句、迭代语句，以及跳转语句共五种。其中，由于前两种语句的展现形式较为动态，因此我着重讲解了语法结构和功能较为固定的后三种语句。

选择语句包含 if…else 与 switch…case 两种类型，编译器通常会采用位映射法、跳表法、基于二分法的测试与条件跳转语句的方式来实现它们。迭代语句则包含 while 语句、do…while 语句以及 for 语句，编译器实现它们时，通常也是采用基本的测试与条件跳转，但三种迭代语句在执行细节上稍有差异。最后的跳转语句包含 return 语句、continue 语句、break 语句以及 goto 语句。它们作为可以改变程序执行顺序的语句，大多也以类似的方式实现。

思考题

最后给你留一个小问题，你可以在留言区和我交流：空语句；在 C 语言中有哪些使用方式？

今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 计算单元：运算符是如何工作的？

下一篇 05 | 代码封装（上）：函数是如何被调用的？

精选留言 (11)

写留言

pedro

2021-12-13

竟然有一种像当年读CSAPP第三章的感觉！

空语句用的少，多数时候用来做空循环体吧。

展开

作者回复: 没错！这是一种常见的场景。举个例子：

```
#include <stdio.h>
int main() {
  char c;
  while ((c = getchar()) == ' ');
  printf("%c", c);
}
```

3

liu_liu
2021-12-15

关于迭代语句中的 .L2 汇编代码段：

// 把 v 放入 eax
mov eax, DWORD PTR [rbp-4]…
展开

作者回复: 这是一个很棒的问题！
如果使用 mov 指令，也是可以完成同样的功能的，只不过需要独立的两条指令，比如：
- dec rax;
- mov edx, rax
但使用 lea 指令，我们就可以仅通过一条机器指令来实现。相对于前者，其执行效率是不是更高呢？实际上，对于编译器来说，很多基本的四则运算都可以直接使用 lea 指令来实现，比如 lea eax, [123 + 4*ebx + esi]，就是直接将 ebx 中的值乘以 4 再加上 123 和 esi 中的值。相较于使用 mov 的多条指令，lea 可以充分利用它在硬件中的电路特性，以更高的效率进行计算。

1

qinsi
2021-12-13

维基上Duff's device的例子

send(to, from, count)
register short *to, *from;
register count;…
展开

作者回复: 达夫设备是 switch 语句和循环展开的巧妙结合，我们会在 18 讲再来深入介绍哈！

1

2021-12-17

像 edi esi 对应的应该是 第一个参数 第二个参数吧 ~ 我记得有 6个还是7个寄存器可以用来传参

dog_brother
2021-12-16

int i = 0;
for(; i <10; ++i) {;}
我经常使用的两种场景，for循环初始条件空，或者循环体为空

送过快递的码农
2021-12-15

老师 ，test/cmp 比较多指令，最终结果会在EFLAGS这个寄存器中么，j(cond)指令会根据这个寄存器做相应的跳动？

展开

作者回复: 没错的，你的理解是正确的。

chinandy
2021-12-14

老师：高优化等级怎么打开，在您给的那个网站上

展开

作者回复: 可以直接在上方的 “Compiler options” 输入框里输入对应优化等级的参数，比如：-O2 \ -O3。

=
2021-12-14

空语句；常用来实现一个空循环。例如:
for(;;);

Geek_828b39
2021-12-14

将寄存器 rdx 中的值左移 v 位（值被扩展为 64 位）；
老师，这个这么理解？

展开

作者回复: 这里实际上是跟蓝框内的第一条汇编语句做的对比。第一条的 mov 指令仅把值 1 当做 32 位值放入到 edx 中；而后续程序在使用这个值生成“token”时，实际上是直接用 rdx 来访问的，因此实际上是在进行左移时同时被“扩展”为了 64 位值。同样的，后面进行按位与时，也是直接以 64 位值进行的。

柴
2021-12-14

我在嵌入式程序里面，写过for(；；;){}这样的死循环语句，其他的没啥印象

共 1 条评论 >

ppm
2021-12-13

延时或在case编译不过时

展开 ∨